Unified Stream Processing Raytracer

Gabriel Moreno-Fortuny ⋆ Michael McCool
Computer Graphics Lab, School of Computer Science, University of Waterloo

Sh is a metaprogramming library that can dynamically generate stream processing code for both GPUs and CPUs [1]. We are extending the semantics of its stream processing model. Simultaneously, we are attempting to target both shared-memory and distributed memory parallel machines in addition to GPUs. The goal is to be able to efficiently run the same algorithm on either a GPU, on a single CPU, or on a parallel machine.

As a test application, we are developing a stream-based implementation of a ray tracer. The goal is to enhance our language to the point that a simple imperative expression of the desired algorithm can be given while still mapping to the multiple targets given above with high efficiency. In addition, we plan to explore the construction of accelerators for dynamic scenes and the use of arbitrary shaders in ray-traced scenes. The latter goal is interesting because it can exploit Sh's capability to dynamically combine code fragments as well as its capability for data-dependent execution within the stream processing model.

Raytracer Testbed

Real time ray tracing has already been demonstrated on both distributed systems [3] and GPUs [2]. However, these implementations were performed at a low level. We are instead using ray tracing as a test case to drive the extension of the Sh language. Our prototype only ray traces static scenes with a fixed Phong lighting model.

We are now exploring extensions that will permit the use of arbitrary shaders on hit surfaces and dynamic scenes. The former extension will exploit the capability of Sh to build large stream programs by dynamically combining several fragments. The second goal will explore the dynamic construction of accelerator data structures. Our goal is a modular ray-tracing system that can itself be used as a component of other systems.

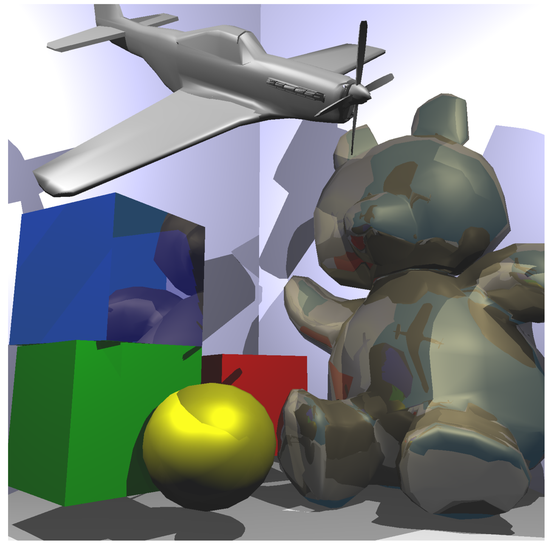
We have developed a ray tracing algorithm that can be expressed using stream computations. We have taken Purcell's approach as a starting point. In our prototype, the following steps are taken to generate a rendered image. First an accelerator structure is created; we use a uniform 3D grid. We then generate rays. A bounding box test detects if each ray actually hits the scene and at which voxel it entered the accelerator grid. Once inside a voxel the program checks for triangle intersections in that voxel. If no hit is found, the voxel traverser advances the ray to the next voxel in the grid. If a triangle is hit, then a shader kernel which calculates the color of the pixel is evaluated.

A simplified version of the stream raytracer was compiled to both a 2.66GHz P4 and a GeForce 6800. The GeForce was approximately 20 times faster.

Tracer Code

Below are the two most important functions of the raytracer. The tracer code shoots light, shadow and reflection rays. The intersector code traverses the accelerator grid and reports hits.

```
ShColor3f RayTracer::tracer(ShPoint2f sc, ShPoint3f eye){
  ShPoint3f inter;
  ShColor3f scolor,color, C;
  scolor = color = ShColor3f(0,0,0);
  ShVector3f V, N, ray;
  ray(0,1)=-sc(0,1); ray(2)=1.0; // create eye ray
  ray=viewMatrix | normalize(ray);
  ShAttrib2f rn = ShConstAttrib2f(1,0);
  ShAttrib1f shadows, r=REFLECTIONS, tracing, finished;
  SH_DO {
    ShColor3f lightcol;
    SH_IF(shadows) { // get light information
      ShPoint3f lightpos;
      getLightValues(srtf(0)-1, lightpos, lightcol);
      ray=normalize(lightpos-inter);      // generate Light vector
      eye=(inter + NEPSILON*N) + DISP*ray;
      shadows-=1; //decrease shadow counter
    } SH_ENDIF;
    ShAttrib1f l, uv, id; // Check for an intersection
    ShAttrib1f hit = intersector(ray, eye , l, uv, id);
    SH_IF(tracing) { // Tracing state
      SH_IF(hit) {
        inter = eye + l*ray;      //store intersection point
        V = normalize(-ray);      // store view vector
        ShTexCoord2f tc = i2tc(id); //access normals, color..
        rn(1) = m_reflect[tc]; //..and reflection coefficients
        ShAttrib3f n0, n1, n2, c0, c1, c2;
        readTables(tc, n0, n1, n2, c0, c1, c2);
        C = barylerp(uv, c0, c1, c2); // linear interpolation
        N = normalize(barylerp(uv, n0, n1, n2));
        tracing = 0; // set state to Shadow
        scolor = ShConstColor3f(0,0,0); //initialize color
        shadows = light_list.size(); //initialize shadow counter
      } SH_ELSE {
          finished = 1; scolor = BACKCOLOR;
      } SH_ENDIF;
    } SH_ELSE { // Shadow State (obtain color)
      scolor+=cond(hit,SHADOW,LIGHT)*shader(V,N,ray,C)*lightcol;
    } SH_ENDIF;
    SH_IF(!srtf(0)) {// Accumulate color and reflection coefficients
      color += rn(0)*scolor;
      rn(0) *= rn(1);
      color -= rn(0)*scolor;
      SH_IF(r && rn(1)) { //..and check for reflections
        ray = reflect(V,N);
        eye = inter + DISP*ray;
        r -= 1; //decrease reflection counter
        tracing = 1; // set state to Tracing
      } SH_ELSE {
        finished = 1; //if no more reflections, finish
      } SH_ENDIF;
    } SH_ENDIF;
  } SH_UNTIL(finished); // finished
  return sat((1 - AMBIENT)*color + AMBIENT);
}
```

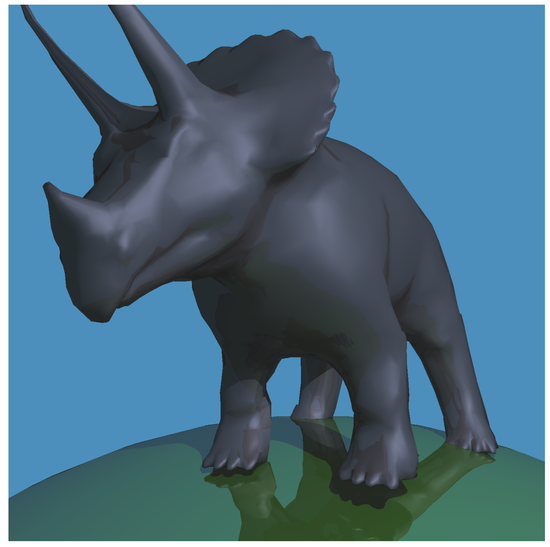


Intersector Code

```
ShAttrib1f RayTracer::intersector(ShVector3f ray,
ShPoint3f eye, ShAttrib1f& l, ShAttrib1f &uv, ShAttrib1f &id){
  ShAttrib3f step, tMax, tDelta;
  ShPoint3f voxel, inter;
  ShAttrib1f hit;
  hit=rayBoxIntersect(ray, eye, inter); // scene bounding box
  SH_IF(hit) {    // if hit, prepare to traverse bounding grid
    voxel = floor((inter - m_grid.gmin)/m_grid.dt);//init voxel
    voxel = pos(min(voxel, fillcast<3>(m_grid.gsize - 1.0)));
    step=cond(ray != ZERO3F, ray/abs(ray), ZERO3F); //step dir
    ShAttrib3f SH_DECL(sptr)= step > ZERO3F; //other vars
    tMax=cond(ray != ZERO3F,
     (m_grid.gmin+(voxel+sptr)*m_grid.dt-inter)/ray,INF3F);
    tDelta=cond(ray!=ZERO3F,m_grid.dt/abs(ray),INF3F);
    ShAttrib1f finished = 0; hit = 0;
    SH_DO {      //main traversing loop
      ShAttrib2f tvptr; // check current voxel
      tvptr(0) = m_grid3D[lookup3d(voxel)];
      ShAttrib2f tindex=cond(tvptr(0)>-1,i2tc(tvptr(0)),ZERO2F);
      tvptr(1)=cond(tvptr(0)>-1,m_triangles[tindex],-ONE1F);
      l = INF1F; // set l to infinity
      SH_WHILE(tvptr(1) > -1) { //check all triangles in voxel
        rayTriangleIntersect(ray, eye, tvptr(1), l, uv, id);
        tvptr(0) += 1;
        tvptr(1) = m_triangles[i2tc(tvptr(0))];
      } SH_ENDWHILE;
      hit = l < INF; //if any triangle is hit, set hit to true
      SH_IF(hit) { // but first make sure its in current voxel
        finished=hit=checkVoxelBound(eye, ray, l, voxel);
      } SH_ENDIF;
      //advance grid traversing variables.
      traverseGrid(tMax, voxel, finished, tDelta, step);
    } SH_UNTIL(finished);
  } SH_ENDIF;
  return hit;
}
```

Conclusion

This algorithm is reasonably efficient and robust but it is dependent on an accelerator structure which must be precomputed. In the future we plan to exploit the fact that in most applications of interest, most of the scene is static with only a small number of dynamic objects. We will use separate accelerators for the static and dynamic parts of the scene, and a simple accelerator (such as a bounding box hierarchy) for the dynamic parts of the scene. Our preliminary mapping of this algorithm to a parallel machine will replicate the scene database and accelerator structure. More efficient and scalable implementations are possible by partitioning the array holding the accelerator structure and sorting stream records to processors by array access addresses.



References

[1] Michael McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Graphics Hardware*, pages 57–68, September 2002.

[2] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *Graphics Hardware*, pages 703–712, 2003.

[3] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray tracing of highly complex models. In *12th Eurographics Workshop on Rendering*, pages 277–288, 2001.